Grammar Fragments Fly First-Class

Marcos Viera[1], Doaitse Swierstra[2], and Atze Dijkstra[2]

[1]Instituto de Computación, Universidad de la República,
Montevideo, Uruguay, mviera@fing.edu.uy
[2]Department of Computer Science, Utrecht University, Utrecht,
The Netherlands, {doaitse,atze}@cs.uu.nl

Abstract

We present a Haskell library for expressing (fragments of) grammars using typed abstract syntax with references. We can analyze and transform such representations and generate parsers from them. What makes our approach special is that we can combine embedded grammar fragments on the fly, i.e. after they have been compiled. Thus grammar fragments have become fully typed, first-class Haskell values.

We show how we can extend an initial, limited grammar embedded in a compiler with new syntactic constructs, either by introducing new non-terminals or by adding new productions for existing non-terminals. We do not impose any restrictions on the individual grammar fragments, nor on the structure as a whole.

1 Introduction

There are many different ways to represent grammars and grammatical structures *in a typeful way*: be it in implicit form using *conventional parser combinators* or more explicitly in the form of *typed abstract syntax*. Each approach has its own advantages and disadvantages. The former, being a domain specific embedded language, makes direct use of the typing, abstraction and naming mechanisms of the host language. This implicit representation however does have its disadvantages: we can only perform a limited form of grammar analysis and transformation since references to non-terminals in the embedded language are implemented by variable references in the host language. The latter approach, which does give us full access to the complete domain specific program, comes with a more elaborate naming system which gives us the possibility to identify references to non-terminals; however transforming such programs in Haskell necessitates to provide proofs (in our case encoded through the Haskell type system) that the types remain correct during transformation.

One of the applications of the latter approach is where one wants to compose grammar fragments. This is needed when a user can extend the syntax of

a base language. In doing so he has to extend the underlying context-free grammar and he has to define the semantics for these new constructs. Once all extensions have become available the parser for the complete language is constructed by joining the newly defined or redefined semantics with the already existing parts. In a more limited way such things can be done by e.g. the quasi quoting mechanism as available through Template Haskell [9, 13], which however has its limitations: the code using the new syntax can be clearly distinguished from the host language. Furthermore the TH code, which is run in a separate phase, is not guaranteed to generate type correct code; only after the code is expanded type checking takes place, often leading to hard to understand error messages.

In an earlier paper [17] we have shown how to define the final semantics of a composed language in terms of composable attribute grammar fragments and in [16] we have shown how to compose grammar fragments for a limited class of grammars, i.e. those describing the output format of Haskell data types. These latter grammars have a convenient property: productions will never derive the empty string, which is a pre-condition for the Left-Corner Transform (LCT) [1] which is to be applied later to remove left-recursion from the grammar which arises from the use of infix data constructors.

In this paper we describe an *unrestricted*, *applicative* interface[1] for constructing such grammar descriptions, we describe how they can be combined, and how they can be transformed so they fulfill the precondition of the LCT. The final result can safely be mapped onto a top-down parser, constructed using a conventional parser combinator library.

In section 2 we describe the "user-interface" to our library. In section 3 we introduce the types used to represent our fragments, whereas in section 4 we describe the internal data structures. In section 5 we discuss some related work and conclude.

2 Context-Free Grammar

In this section we show how to express a context free grammar fragment. Our running example is a simple expression language (see Figure 1, the *initial language*). Note that this concrete grammar uses the syntactic categories *root*, *exp*, *term* and *factor* to express operator precedences.

Figure 1 also shows the almost isomorphic Haskell code encoding of this language fragment in terms of our combinator library and the *Arrow*-interface [7, 11]. A grammar description is an *Arrow*, representing the introduction of its composing non-terminals. Since non-terminals can be mutually recursive, they are declared into a **rec** block. The function *addNT* introduces a new non-terminal together with some initial productions (alternatives) separated by `<|>` operators. Each alternative (right hand side of a production) consists of a sequence of elements, expressed in so-called *applicative style*, using the

[1] Available at: `http://hackage.haskell.org/package/SyntaxMacros`

```
Grammar:

  root   ::= exp
  exp    ::= "let" var "=" exp "in" exp | exp "+" term | term
  term   ::= term "*" factor | factor
  factor ::= int | var

Haskell code:

  prds = proc () → do
    rec root   ← addNT ≺ ⟦ semRoot exp ⟧
        exp    ← addNT ≺ ⟦ semLet  "let" var "=" exp "in" exp ⟧
                       <|> ⟦ semAdd  exp  "+" term  ⟧ <|> ⟦ id term  ⟧
        term   ← addNT ≺ ⟦ semMul  term "*" factor ⟧ <|> ⟦ id factor ⟧
        factor ← addNT ≺ ⟦ semCst  int ⟧ <|> ⟦ semVar var ⟧
    exportNTs ≺ exportList root $ export ntExp    exp
                                  . export ntTerm   term
                                  . export ntFactor factor
  gram = closeGram prds
```

Figure 1: Initial language

idiomatic brackets[2] ⟦ and ⟧ which delineate the description of a production from the rest of the Haskell code. The brackets ⟦ and ⟧ are syntactic sugar for the Haskell function *iI* and constant *Ii*. A production consists of a call to a *semantic function*, which maps the results of the trailing non-terminals to the result of this production, and a sequence of non-terminals and terminals, the latter corresponding to literals which are to be recognized. Since terminal symbols like `"let"` and `'*'` do not bear any meaning our idioms automatically discard these results: the expression ⟦ *semMul term* `"*"` *factor* ⟧ is equivalent to *pure* $(\lambda l \ \_ \ r \rightarrow semMul\ l\ r)$ `<*>` *sym term* `<*>` *tr* `"*"` `<*>` *sym factor* in the *Applicative* interface [10]. The semantic functions are defined elsewhere (using monad transformers, attribute grammars or embedded AG code [17]). By convention we will let their names start with *sem*. For elementary parsers which return values which are constructed by the scanner we provide a couple of predefined special cases, such as *int* which returns the integer value from the input and *var* which returns a recognized variable name.

An initial grammar is also an *extensible grammar*. It exports (with *exportNTs*) its starting point (*root*) and a list of *exportable non-terminals* which actually stand for a collection of productions to be used and modified in future extensions. Each *export*-ed non-terminal is labeled by a unique value of a unique type (by convention starting with *nt*). The function *closeGram* takes the list

[2] `http://www.haskell.org/haskellwiki/Idiom_brackets`

of productions, and converts it into a compiler; in our case a parser integrated with the semantics for the language derived from the starting symbol *root*.

2.1 Language Extension

We now extend the language with an extra non-terminal for conditions (Boolean expressions) and extra productions for conditional expressions and parentheses:

```
exp    ::= ...              | "if" cond "then" exp "else" exp
cond   ::= exp "==" exp     | exp ">" exp
factor ::= ...              | "(" exp ")"
```

This language extension *prds′* is defined as a closed Haskell value by itself, which accesses an already existing set of productions (*imported*) and builds an extended set, as shown in Figure 2.

```
prds' = proc imported → do
  let exp    = getNT ntExp    imported
  let factor = getNT ntFactor imported
  rec addProds ≺ (exp,    ⟦ semIf   "if" cond "then" exp "else" exp ⟧)
      cond ← addNT ≺      ⟦ semEq exp "==" exp ⟧ <|> ⟦ semGr exp ">" exp ⟧
      addProds ≺ (factor, ⟦ semPar "(" exp ")" ⟧)
  exportNTs ≺ extendExport imported (export ntCond cond)
gram' = closeGram (prds +>> prds')
```

Figure 2: Language Extension

For each non-terminal to be extended we retrieve its current list of productions (using *getNT*) from the *imported* non-terminals, and add new productions to this list using *addProds*. The **if**-expression is e.g. added by:

```
let exp = getNT ntExp imported
addProds ≺ (exp, ⟦ semIf "if" cond "then" exp "else" exp ⟧)
```

New non-terminals can be added as well using *addNT*; in the example we add the non-terminal *cond*:

```
cond ← addNT ≺ ⟦ semEq exp "==" exp ⟧ <|> ⟦ semGr exp ">" exp ⟧
```

Finally, we extend the list of exportable non-terminals with (some of) the newly added non-terminals, so they can be extended by further fragments elsewhere:

```
exportNTs ≺ extendExport imported (export ntCond cond)
```

The original grammar *prds* is extended with *prds′* using the combinator (+>>). Because both *prds* and *prds′* are proper Haskell values which can be separately defined in different modules and compiled separately we claim that the term *first class grammar fragments* is justified here. It is important to note that all these productions describe well-typed Haskell values, of which the type is parameterized with the type of values the expressions represent; so, based on the type of *semIf* the Haskell compiler will be able to check that the non-terminal *cond* indeed parses Boolean expressions! By being able to compile a language and its extensions separately, a framework for extensible compilers can be defined which allows a language to be extended without providing full access to its source and without having to re-compile the whole compiler.

3 Grammar Representation

Having described how a user describes and combines individual language fragments, we now embark on the description of the internals of our library.

One of the basic requirements we want to fulfill is that components can be safely composed in all circumstances, without imposing strong requirements on individual components, since this would soon make the system useless. So we will have to deal with left-recursive grammars and grammars which are, once composed, for example not LALR(1). In previous work [3, 1, 16] we have developed a series of techniques to deal with such grammars, all based on *typed representations* and *typed transformations* of grammars, for example to remove left recursion. In this section we introduce a typed representation of grammars that *provides an easy way to describe grammars* and enables the use of these techniques.

We represent grammars as typed abstract syntax, using Generalized Algebraic Data Types [12] (GADTs). The idea, proposed in [1], is to indirectly refer to non-terminals via references encoded as types. Such references type-index into an environment holding the actual descriptions of the non-terminals.

A *Ref* encodes a typed index into an environment containing values of different types. It is labeled with the type a associated with the referenced value and the type *env* of an environment (a nested Cartesian product extending to the right) into which the reference indexes. The constructor *Zero* expresses that the first element of the environment has to be of type a. The constructor *Suc* remembers a position in the rest of the environment. It ignores the first element in the environment by being polymorphic in the type b:

```
data Ref a env where
  Zero ::                    Ref a (env', a)
  Suc  :: Ref a env' →       Ref a (env', b)
data Env t use def where
  Empty :: Env t use ()
  Ext   :: Env t use def' → t a use → Env t use (def', a)
```

Since we want to be able represent mutually recursive definitions, an environment *Env* contains terms (in our case productions) which may contain typed references to other terms: the type of such a term is $t\ a\ use$, where t describes the kind of terms we store, the type parameter a is the type of the value described by the term and *use* is the type labeling the environment into which references to other terms occurring in the term may point. The type parameter *def* contains the type labels a of the terms of type $t\ a\ use$ defined by the environment: whenever the constructor *Ext* is used to extend and environment $Ent\ t\ use\ def'$ with a term $t\ a\ use$ the type label of the resulting environment is extended with this $a\ (def', a)$.

A type *FinalEnv* forces environments *def* and *use* to coincide, thus making sure that all references point to some definition, and that those definitions describe values of the appropriate types.

```
type FinalEnv t usedef = Env t usedef usedef
```

A *Grammar* consists of a closed environment, containing a list of productions for each non-terminal, and a reference $(Ref\ a\ env)$ to the root symbol, where a is the type of the witness of a successful parse. Note that the type *env* is hidden using existential quantification, so changes to the structure of the grammar can be made, by adding or removing non-terminals, without having to change the visible part of its type.

```
data EG; data CG
data Grammar s a = ∀ env.Grammar (Ref a env)
                                 (FinalEnv (Productions s) env)
newtype Productions s a env = PS { unPS :: [Prod s a env] }
```

The type s encodes the state of the grammar, that is: *EG* if the grammar can contain empty productions and *CG* if the grammar does not.

For productions we choose a representation which differs slightly from the one used in [1, 16]. Here we represent productions in an *applicative-style*; i.e. using a couple of constructors *Pure* and *Seq* analogous to the *pure* function and `<*>` operator of applicative functors:

```
data Prod s a env where
  Pure :: a                    → Prod s a env
  Seq  :: Prod s (a → b) env
       → Prod s a         env → Prod s b env
  Sym  :: Symbol a t env       → Prod s a env
```

Sym is a special case of *pure* that lifts a symbol to a production. A symbol is either a terminal or a non-terminal encoded by a reference pointing to one of the elements in an environment labeled with *env*. A normal terminal contains the literal string it represents. We define a category of *attributed terminals*, which are not fixed by a literal string. Every attributed terminal refers to a lexical structure. In contrast to the normal terminals which do not bear a

semantic value, for attributed terminals the parsed values are used and the type a instantiates to the type of the parsed value.

```
data TTerm; data TNonT; data TAttT
data Symbol a t env where
  Term       :: String →      Symbol String TTerm  env
  Nont       :: Ref a env → Symbol a      TNonT env
  TermInt    ::               Symbol Int   TAttT  env
  TermVarid  ::               Symbol String TAttT  env
```

The type parameter t indicates, at the type-level, whether a *Symbol* is a terminal (type *TTerm*) for which the result is (usually) discarded, a non-terminal (*TNonT*) or an attributed terminal (*TAttT*) in the value of which we are interested. In order to make our code more readable we introduce the smart constructors *trm*, *int* and *var*, for the terminals *Term*, *TermInt* and *TermVarid*, respectively.

3.1 From Grammar to Parser

A grammar can be compiled into a top-down parser, which can then be used to *parse* a String into a *ParseResult* containing a semantic value of type a:

```
compile :: Grammar CG a → Parser a
parse   :: Parser a → String → ParseResult a
```

We translate to the `uu-parsinglib` (or any other) parser combinator library [14], that has an *Applicative* (and *Alternative*) interface. Thus, *compile* translates a *Productions* list as a sequence of parsers combined by `<|>`. The *Prod* constructors *Seq* and *Pure* are translated to `<*>` and *pure*, respectively. Terminals are translated to terminal parsers and non-terminal references are retrieved from an environment containing the translated productions for each non-terminal. Notice that only *CG* grammars can be compiled.

3.2 Applicative Interface

We want the type *Productions* to be an instance of the Haskell classes *Applicative* and *Alternative* themselves as we have seen in the examples. However, this is impossible due to the order of its type parameters; we need a to be the last parameter[3]. Thus, we define the type *PreProductions* for descriptions of (possibly empty) alternative productions.

```
newtype PreProductions env a = PP { unPP :: [Prod EG a env] }
```

The translation from *PreProductions* to *Productions* is trivial:

[3] We cannot just redefine *Productions* with this order, because we need the current order for the transformations we will introduce later.

```
prod :: PreProductions env a → Productions EG a env
prod (PP ps) = PS ps
```

Now we can define the (*PreProductions env*) instances of *Applicative* and *Alternative*:

```
instance Applicative (PreProductions env) where
  pure f = PP [Pure f]
  (PP f) <*> (PP g) = PP [Seq f' g' | f' ← f, g' ← g]
instance Alternative (PreProductions env) where
  empty = PP []
  (PP f) <|> (PP g) = PP (f ++ g)
```

We are dealing with lists of alternative productions, thus the alternative operator (`<|>`) takes two lists of alternatives and just appends them. In the case of sequential application (`<*>`) a list of productions is generated with all the possible combinations of the operands joined with a *Seq*.

We also defined smart constructors for symbols: *sym* for the general case and *tr* for the special case where the symbol is a terminal.

```
sym  :: Symbol a t env → PreProductions env a
sym s = PP [Sym s]
tr   :: String → PreProductions env String
tr   s = sym (Term s)
```

4 Extensible Grammars

In this section we present our approach to define and combine *extensible grammars* (like the one in Figure 1) and *grammar extensions* (Figure 2). The key idea is to see the definition, and possibly future extensions, of a grammar as a typed transformation that introduces new non-terminals into a typed grammar.

4.1 TTTAS

Grammar definitions and extensions are defined as typed transformations of values of type *Grammar*. For example, both *prds* and *prds′* of Figures 1 and 2 are typed transformations: while *prd* starts with an empty context-free grammar and transforms it by adding the non-terminals *root*, *exp*, *term* and *factor*, the grammar extension *prd′* continues the transformation started by *prd* and modifies the definition of some of the non-terminals and adds some new ones. Notice that a *Grammar* is a collection of mutually recursive typed structures; thus, performing transformations while maintaining the whole collection well-typed is non-trivial. The rest of this sub-section is a short introduction to the API of TTTAS[4] (Typed Transformations of Typed Abstract Syntax), the library we use to implement our transformations. TTTAS is based on the *Arrow*

[4] `http://hackage.haskell.org/package/TTTAS`

type *Trafo*, which represents typed transformation steps, (possibly) extending an environment *Env*:

$$\textbf{data}\ \mathit{Trafo}\ m\ t\ s\ a\ b$$

The arguments are the types of: the meta-data m (i.e., state other than the environment we are constructing), the terms t stored in the environment, the final environment s, the arrow-input a and arrow-output b. Thus, instances of the classes *Category* and *Arrow* are implemented for $(\mathit{Trafo}\ m\ t\ s)$, which provides a set of functions for constructing and combining *Trafo*s. Some of these functions which we will refer to are:

Identity arrow: $\mathit{returnA} :: \mathit{Arrow}\ a \Rightarrow a\ b\ b$

Lifting functions: $\mathit{arr} :: \mathit{Arrow}\ a \Rightarrow (b \to c) \to a\ b\ c$

Left-to-right comp.: $(>>>) :: \mathit{Category}\ \mathit{cat} \Rightarrow \mathit{cat}\ a\ b \to \mathit{cat}\ b\ c \to \mathit{cat}\ a\ c$

The class *ArrowLoop* is instantiated to provide feedback loops with:

$$\mathit{loop} :: \mathit{ArrowLoop}\ a \Rightarrow a\ (b, d)\ (c, d) \to a\ b\ c$$

A transformation is run with *runTrafo*, starting with an empty environment and an initial value of type a. The universal quantification over the type s ensures that transformation steps cannot make any assumptions about the type of the (yet unknown) final environment.

$$\mathit{runTrafo} :: (\forall\ s.\mathit{Trafo}\ m\ t\ s\ a\ (b\ s)) \to m\ () \to a \to \mathit{Result}\ m\ t\ b$$

The result of running a transformation is encoded by the type *Result*, containing the final meta-data, the output type and the final environment. It is existential in the final environment, because in general we do not know how many definitions will be introduced by a transformation and which are their associated types. Note that the final environment has to be closed (hence the use of *FinalEnv*).

$$\textbf{data}\ \mathit{Result}\ m\ t\ b = \forall\ s.\mathit{Result}\ (m\ s)\ (b\ s)\ (\mathit{FinalEnv}\ t\ s)$$

New terms can be added to the environment by using the function *newSRef*. It takes the term of type $t\ a\ s$ to be added as input and yields as output a reference of type $\mathit{Ref}\ a\ s$ that points to this term in the final environment:

$$\mathit{newSRef} :: \mathit{Trafo}\ \mathit{Unit}\ t\ s\ (t\ a\ s)\ (\mathit{Ref}\ a\ s)$$
$$\textbf{data}\ \mathit{Unit}\ s = \mathit{Unit}$$

The type *Unit* is used to express the fact that this transformation does not record any meta-information.

Functions of type $(\mathit{FinalEnv}\ t\ s \to \mathit{FinalEnv}\ t\ s)$ which update the final environment of a transformation can be lifted into the *Trafo* and composed using

updateFinalEnv. All functions lifted using *updateFinalEnv* will be applied to the final environment once it is created.

```
updateFinalEnv :: Trafo m t s (FinalEnv t s → FinalEnv t s) ()
```

If we have, for example:

```
proc () → do updateFinalEnv ≺ upd1
             ...
             updateFinalEnv ≺ upd2
```

the function (*upd2*.*upd1*) will be applied to the final environment, produced by the transformation.

4.2 Grammar Extensions

In this subsection we present the API of a library for defining and combining *extensible grammars* (like the one in Figure 1) and *grammar extensions* (Figure 2).

A grammar extension can be seen as a series of typed transformation steps that can add new non-terminals to a typed grammar and/or modify the definition of already existing non-terminals. We define an extensible grammar type (*ExtGram*) for constructing initial grammars from scratch and a grammar extension type (*GramExt*) as a typed transformation that extends a typed extensible grammar. In both cases a *Trafo* uses the *Productions* as the type of terms defined in the environment being carried.

```
type ExtGramTrafo = Trafo Unit (Productions EG)
type ExtGram env            start' nts'
   = ExtGramTrafo env ()                    (Export start' nts' env)
type GramExt env start nts start' nts'
   = ExtGramTrafo env (Export start nts env) (Export start' nts' env)
```

4.2.1 Exportable non-terminals

Both extensible grammars and grammar extensions have to export the starting point *start'* and a list of *exportable non-terminals nts'* to be used in future extensions. The only difference between them is that a grammar extension has to import the elements (*start* and *nts*) exported by the grammar it is about to extend, whereas an extensible grammar, given that it is an initial grammar, does not need to import anything.

The exported (and imported, in the case of grammar extensions) elements have type *Export start nts env*, including the starting point (a non-terminal, with type *Symbol start TNonT env*) and the list of exportable non-terminals (*nts env*).

```
data Export start nts env = Export (Symbol start TNonT env) (nts env)
```

The list of exportable non-terminals has to be passed in a *NTRecord*, which is an implementation of extensible records very similar to the one in the *HList* library [8], with the difference that it has a type parameter *env* for the environment where the non-terminals point into. A field $(l \in v)$ relates a (first-class) non-terminal label l with a value v. A *NTRecord* can be constructed with the functions (`.*.`), for record extension, and *ntNil*, for empty records. The function *getNT* is used to retrieve the value part corresponding to a specific non-terminal label from a record. We have defined some functions to construct *Export* values:

```
exportList r ext = Export r (ext ntNil)
export l nt      = (.*.) (l ∈ nt)
```

Thus, the export list in Figure 1 is equivalent to:

```
Export root (ntExp ∈ exp .*. ntTerm ∈ term .*. ntFactor ∈ factor .*. ntNil)
```

In order to finally export the starting point and the exportable non-terminals we chain an *Export* value through the transformation in order to return it as output.

```
exportNTs :: NTRecord (nts env)
          ⇒ ExtGramTrafo env (Export start nts env) (Export start nts env)
exportNTs = returnA
```

Thus, the definition of an extensible grammar (like the one in Figure 1) has the following shape, where *exported_nts* is a value of type *Export*:

```
prds = proc () → do {...; exportNTs ≺ exported_nts}
```

The definition of a grammar extension, like the one in Figure 2, has the shape:

```
prds' = proc (imported_nts) → do {...; exportNTs ≺ exported_nts}
```

where *imported_nts* and *exported_nts* are both of type *Export*. We have defined a function to extend (imported) exportable lists:

```
extendExport (Export r nts) ext = Export r (ext nts)
```

4.2.2 Adding Non-terminals

To add a new non-terminal to the grammar we need to add a new term to the environment.

```
addNT :: ExtGramTrafo env (PreProductions env a) (Symbol a TNonT env)
addNT = proc p → do {r ← newSRef ≺ prod p; returnA ≺ Nont r}
```

The input to *addNT* is the initial list of alternative productions (*PreProductions*) for the non-terminal and the output is a non-terminal symbol, i. e. a reference to the non-terminal in the grammar. Thus, when in Figure 1 we write:

$$term \leftarrow addNT \prec [\![\ semMul\ term\ \texttt{"*"}\ factor\ ]\!]\ \texttt{<|>}\ [\![\ id\ factor\ ]\!]$$

we are adding the non-terminal for terms, with the alternative productions $[\![\ semMul\ term\ \texttt{"*"}\ factor\ ]\!]$ and $[\![\ id\ factor\ ]\!]$, and we bind to *term* a symbol holding the reference to the added non-terminal thus making it available to be used in the definition of this or other non-terminals. Because *Trafo* instantiates *ArrowLoop*, we can define mutually recursive non-terminals using the keyword **rec**, like in figures 1 and 2.

4.2.3 Adding (and Removing) Productions

Adding new productions to an existing non-terminal translates into the concatenation of the new productions to the existing list of productions of the non-terminal.

```
addProds :: ExtGramTrafo env
                    (Symbol a TNonT env, PreProductions env a) ()
addProds = proc (Nont r, prds) → do
  updateFinalEnv ≺ updateEnv (λps → PS $ (unPP prds) ++ (unPS ps)) r
```

In Figure 2 we have seen examples of adding productions to the non-terminals *exp* and *factor*. It is easy to define a function *remProds*, to remove all the productions of a non-terminal, based on the code of *addProds*. The difference is that the function which updates the environment is now $(\backslash\_ \rightarrow PS\ [\,])$. Note that removing the non-terminal completely would be much harder.

4.2.4 Grammar Extension and Composition

Extending a grammar boils down to composing two transformations, the first one constructing an extensible grammar and the second one representing a grammar extension.

```
(+>>) :: (NTRecord (nts env), NTRecord (nts' env))
      ⇒ ExtGram env start  nts → GramExt env start nts start' nts'
      → ExtGram env start' nts'
g +>> sm = g >>> sm
```

We defined (+>>) to restrict the types of the composition. Two grammar extensions can be composed just by using the (>>>) operator from the *Arrow* class.

If we want to compose two extensible grammars *g1* and *g2* with disjoint non-terminals sets, we have to sequence them, obtain their start points *s1* and *s2* and add a new starting point *s* with *s1* and *s2* as productions.

```
(<++>) :: (NTUnion nts1 nts2 nts)
       ⇒ ExtGram env start nts1 → ExtGram env start nts2
       → ExtGram env start nts
```

```
g1 <++> g2 = proc () → do (Export s1 ns1) ← g1 ≺ ()
                          (Export s2 ns2) ← g2 ≺ ()
                          s ← addNT ≺ sym s1 <|> sym s2
                          returnA ≺ Export s (ntUnion ns1 ns2)
```

The function *ntUnion* performs the union of the non-terminal labels both at value and type level. It introduces the constraint *NTUnion*, which also ensures the disjointedness of the sets. If we have a function *ntIntersection*, returning for each intersecting non-terminal its position on each grammar $(nt1, nt2)$, then we can define a general composition of grammars. We have to extend (<++>) with the transformation $(addProds\ (nt1, \llbracket\ id\ nt2\ \rrbracket)$ >>> $addProds\ (nt2, \llbracket\ id\ nt1\ \rrbracket))$ for each non-terminal belonging to the intersection.

4.3 Closed Grammars

To close a grammar we run the *Trafo*, in order to obtain the grammar to which we apply the Left-Corner Transform. By applying *leftcorner* we prevent the resulting grammar to be left-recursive, so it can be parsed by a top-down parser. Such a step is essential since we cannot expect from a large collection of language fragments, that the resulting grammar will be e.g. LALR(1) or non-left-recursive. The type of the start non-terminal a is the type of the resulting grammar.

```
closeGram :: (∀ env.ExtGram env a nts) → Grammar CG a
closeGram prds = case runTrafo prds Unit () of
  Result _ (Export (Nont r) _) gram
    → (leftCorner.removeEmpties) (Grammar r gram)
```

The *leftcorner* function is an adaptation to our representation of *Prod* of the transformation proposed in [2] which preprocesses the grammar such that empty parts at the beginning of productions have been removed:

```
removeEmpties :: Grammar EG a → Grammar CG a
leftCorner    :: Grammar CG a → Grammar CG a
```

The function *removeEmpties* takes a grammar that can have empty productions (*Grammar EG a*) and returns an equivalent grammar (*Grammar CG a*) without empty productions and without left-most empty elements. In a companion technical report [15] we provide a complete implementation of the transformation, which basically consists of removing the empty production of each non-terminal and adding it to the contexts where the non-terminal is referenced. Thus, if the root symbol has an empty production, allowing the parsing of the empty string, this behavior will not be present after the removal. For simplicity reasons we avoid this situation by disallowing empty productions for the root symbol of the grammars we deal with. It is easy to remove this constraint by re-adding the empty production to the start non-terminal of the grammar resulting from the whole (*leftCorner.removeEmpties*) transformation. But in practice we do not expect this to be necessary.

5 Related Work and Conclusions

This paper builds on our previous work on typed transformations of typed grammars [2, 1, 16], although we here stuck more to the conventional applicative style in order to make it more accessible to the everyday programmer who knows Haskell. The major contribution of this paper is the introduction of a set of combinators to describe, extend and combine grammar fragments using arrow notation. In order to avoid problems with the constructed grammars we have introduced a preprocessing step before applying the LCT.

Of course there exist a myriad of other approaches to represent context-free grammars and grammar fragments, but we are not aware of the existence of a *typeful way of representing grammar fragments using an embedded domain specific language* as we have presented here. Because of the embeddedness it remains possible to define one's own grammar constructs such as sequences, optional elements and chains of elements, thus keeping all the advantages commonly found in combinator parser based approaches.

Devriese and Piessens [6] propose a model for explicitly recursive grammars in Haskell, which provides an applicative interface to describe productions. By using generic programming techniques from [18] their representation supports a wide range of grammar algorithms, including the Left-Corner Transform.

Brink et al. [5] introduced a framework to represent grammars and grammar transformations in the dependently typed programming language Agda. In such a language they are able to prove correctness properties of the transformations, more than the preservation of semantic types.

On one hand both claim to be less complex than our technique, but on the other hand, they are both based on closed non-terminal domains, and thus they lack of grammar extension and composition which form the core of this paper.

Finally note that the way in which we eventually construct parsers out of the constructed grammar in no way precludes other approaches. So it is a trivial extension to generate parsers which can deal with ambiguous grammars by using suitable combinators (like the *amb* combinator from the `uu-parsinglib` library). If one wants to use very general parsing techniques or scannerless parsing techniques [4] there is nothing that prevents one from doing so. No information is lost in the final representation.

References

[1] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed grammars: The left corner transform. *Electron. Notes Theor. Comput. Sci.*, 253:51–64, September 2010.

[2] Arthur Baars, S. Doaitse Swierstra, and Marcos Viera. Typed transformations of typed abstract syntax. In *TLDI 2009*, pages 15–26.

[3] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP 2002*, pages 157–166.

[4] Martin Bravenboer. *Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* PhD thesis, Utrecht University, Utrecht, The Netherlands, January 2008.

[5] Kasper Brink, Stefan Holdermans, and Andres Löh. Dependently typed grammars. In *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 58–79. 2010.

[6] Dominique Devriese and Frank Piessens. Explicitly recursive grammar combinators: a better model for shallow parser dsls. In *PADL 2011*, pages 84–98.

[7] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[8] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004*, pages 96–107.

[9] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell 2007*, pages 73–82, 2007.

[10] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.

[11] Ross Paterson. A new notation for arrows. In *ICFP 2001*, pages 229–240.

[12] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. *SIGPLAN Not.*, 41(9):50–61, 2006.

[13] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[14] S. Doaitse Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, , and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Spinger, 2009.

[15] Marcos Viera, S. Doaitse Swierstra, and Atze Dijkstra. Grammar fragments fly first-class. UU-CS 032, Utrecht University, 2011.

[16] Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell 2008*, pages 63–74.

[17] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: how to do aspect oriented programming in haskell. In *ICFP 2009*, pages 245–256.

[18] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP 2009*, pages 233–244.